

Rooting Around

Integer Square Roots on Small Processors

There are numerous ways to compute square roots. Have you ever tried the sum-of-odds method? What about the bisection method? Michael goes over the basics and shows you how to choose algorithms to suit your particular applications.

Square roots and I go way back. I was in high school when so-called "four-banger" calculators became affordable. These calculators had only four standard functions: addition, subtraction, multiplication, and division. But what if I wanted to do more exotic computations like square root? My local bookstore had several books about calculator tricks, so I picked up a couple to see what I could do with my new toy. (Remember the excitement of punching in "07734" and turning the calculator upside-down to read the "HELLO" message?) One of these books described the Newton-Raphson method for computing square roots, which is an iterative procedure that used only the standard four functions. I memorized the algorithm, and could still punch it out long after I had forgotten the tedious high school method for computing square roots by hand.

My interest in the process of computing square roots was renewed a decade ago when I read an exchange on the *Circuit Cellar* BBS. Somebody asked about the sum-of-odds method, which is a simple and seemingly fast way to compute integer square roots. *Circuit Cellar* columnist Ed Nisley suggested the use of bisection (or binary search) to quickly find the square root using a processor with hardware multiply.

Over the years, I returned to the square root problem on numerous occasions. I came up with several algorithms that I later learned were well known in the field. In this article, I'll describe how these algorithms work.

I'll also help you determine which algorithm is best for your application.

WHY ROOT?

Before getting into the details, let's talk about why you'd want to calculate a square root. As an embedded programmer, I frequently use a square root to determine the root mean square (RMS) value of an AC voltage. This involves taking a square root.

Other applications that use square roots are graphic algorithms and robot navigation systems. For example, to compute the distance between two points on a plane, you must use the Pythagorean theorem:

$$d = \sqrt{\Delta x^2 + \Delta y^2}$$

So, why not just use the square root function in the C standard library? The `sqrt()` C function computes the square root of a double-precision floating-point number, and returns a result of the same type. If all you need is the integer square root of a 16- or 32-bit integer, the overhead of the floating-point operations will slow down your application. For example, with the GCC compiler for AVR processors, `sqrt(50000)` requires 2,254 cycles. The optimized 16-bit shift-subtract algorithm presented here takes only 166 cycles, which is a 13× improvement.

The wrong integer algorithm can also slow down your application. As demonstrated in this article, the difference between the slowest and fastest algorithms is an order of magnitude or more. Using commercial code doesn't guaran-

tee an efficient algorithm. In fact, one of the commercial AVR compilers has integer square root functions that use an inefficient (though highly optimized) exhaustive-search algorithm. You might expect integer functions to be faster than floating point, but in my test of this compiler, taking the square root of 4,294,967,295 (the largest unsigned 32-bit integer) took approximately 655,000 cycles for the 32-bit integer function, 14,700 cycles for the floating-point function, and a mere 670 cycles for a 32-bit implementation of the shift-subtract algorithm presented later in this article. That's almost a 1,000× difference between the slowest and fastest algorithms!

The main topic in this article is integer square roots of integer arguments. All of the functions presented here take a 16-bit unsigned argument *A* and return a 16-bit unsigned integer result: $x = \sqrt{A}$.

Are you wondering why the return value is 16 bits instead of 8 bits? The width of the result is only half that of the argument, except for the routine that rounds the result instead of truncating. The square root of 65,281 (or higher) rounds to 256, which requires 9 bits. In order to have a common interface, I used only 16-bit return values.

EXHAUSTIVE SEARCH

I'll start with the exhaustive-search method, which is the simplest method. An exhaustive search basically tries every possible result until it finds the correct answer. Figure 1a tests succes-

...values of x , starting at 1 and continuing to 255. For each loop, it compares x^2 with the argument A . If x^2 is greater than A , it terminates. (The test for $x = 0$ handles the case, where x rolls over from 255 to 0 because of the

use of an 8-bit counter.) When the loop terminates, you know that the value of x is one greater than $\sqrt{A}$. The function returns $x - 1$.

Although simple, this algorithm has two drawbacks. First, it requires a

multiplication every loop, which may or may not be a disadvantage, depending on whether or not your processor has a hardware multiply instruction. Second, like the other exhaustive-search algorithms, it loops $\sqrt{A}$ times. For large values of A , this can be significant, especially when the algorithm is extended to 32-bit arguments.

Another exhaustive-search algorithm that eliminates the multiplication is the sum-of-odds method. In an exhaustive search, you need to know the value of x^2 for $x = 1, 2, 3, \dots$. Because x is increasing by 1 each loop, there's a shortcut to determine x^2 without performing a multiplication.

Suppose you know the value of x^2 and you want to find the value of $(x + 1)^2$. Multiplying this out, you get:

$$(x + 1)^2 = x^2 + 2x + 1$$

But $2x + 1$ is just the index of successive odd numbers (i.e., first $x + 1$ odd number). ($2x$ gives successive even numbers. Adding 1 gives odd numbers.)

a) <pre>uint16_t exh_multiply (uint16_t A) { uint8_t x = 1; while (((uint16_t)x * x) <= A) { ++x; if (x == 0) { break; } } --x; return (x); }</pre>	b) <pre>uint16_t exh_sumodd (uint16_t A) { uint16_t x_sq = 1; uint16_t x_odd = 1; while (x_sq <= A) { x_odd += 2; x_sq += x_odd; if (x_sq == 0) { break; } } return (x_odd >> 1); }</pre>	c) <pre>uint16_t exh_sumodd_opt (uint16_t A) { uint16_t x_odd = 1; while (x_odd <= A) { A -= x_odd; x_odd += 2; } return (x_odd >> 1); }</pre>
---	---	--

Figure 1—These functions use an exhaustive-search algorithm to calculate the square root of a 16-bit number. Exhaustive-search algorithms try all possible solutions until they find the correct solution. The simplest algorithm uses multiplication to compute the square of each trial (a). The sum-of-odds method computes the square of each successive trial (b). An optimized sum-of-odds method improves performance (c). Note that `uint8_t` and `uint16_t` are type definitions for unsigned 8- and 16-bit integers.

Micro-Server Internet Appliance Engine

SOM-5282EM

- Coldfire 66/80 Mhz CPU
- 10/100BaseT Fast Ethernet
- 16 GP I/Os & 8 Channel A/D
- Programmable in Java™ or C
- Telnet, FTP, and HTTP Servers
- 3 Serial Ports, CAN 2.0B & SPI
- uClinux with Real Time support
- 4.5 MB Flash & up to 16 MB RAM
- Typical Power Consumption < 2 Watts
- Real Time Clock & Nonvolatile Memory
- Carrier/Socket Board & Power Supply Available
- Small, 144 pin SoDIMM form factor (2.66" x 1.5")

The SoM-5282EM is a System on a Module, based on the Freescale MCF5282 Processor. This 32-Bit processor runs uClinux making it extremely easy to create smart Network/Internet capable devices, with Data Acquisition and Control properties. If Real-Time processing is required we can optionally provide RTAI Real-Time extensions. Write sophisticated network applications in days instead of months using standard GNU tools. Single Unit pricing starts at \$150. Optional Carrier/Socket board, Enclosure, and Power Supply also available. For additional information visit, www.emacinc.com/som/.

Since 1985
OVER
20
YEARS OF
SINGLE BOARD
SOLUTIONS

EMAC, inc.
EQUIPMENT MONITOR AND CONTROL

Phone: (618) 529-4525 • Fax: (618) 457-0110 • Web: www.emacinc.com

Tianma LCD's



Tianma Microelectronics specializes in the manufacture of all LCD technologies from TN to TFT. With more than 22 years of LCD experience we have grown to be China's largest LCD manufacturer. Tianma has teams of dedicated and experienced engineers and sales people worldwide to help you with all of your LCD needs. Our continued success in cell phone, automotive, consumer, and medical industries assure our ability to help you in whatever market you serve.

TIANMA

21138 Commerce Pointe Drive Walnut, CA 91789 USA
Tel: (909) 468-2839 Fax: (909) 468-0868
Email: sales@tianma.com

Figure 1b shows the sum-of-odds algorithm in action. This function doesn't store the value of x . Instead it keeps track of x^2 and the x^{th} odd number (in x_sq and x_odd). After each loop, it advances to the next odd number by adding two. It then determines x^2 by adding the odd number to the previous x^2 . Just like Figure 1a, it compares x^2 with A , but it does so without multiplication.

After the loop terminates, the function needs to return x . But because it hasn't stored x in a variable, it must calculate it. Because $x_odd = 2x + 1$, in order to convert x_odd to x , divide by 2 and truncate.

Figure 1c shows an optimized version of the sum-of-odds algorithm. Instead of keeping track separately of x^2 and comparing with A , this function subtracts the odd numbers directly from A . Because of this, the variable A is always the difference between the original argument and x^2 . When that difference goes negative, the loop terminates.

As a final implementation of the sum-of-odds algorithm, I wrote an optimized function in AVR assembly for the Atmel ATmega8 processor. Not surprisingly, this optimized assembly code is the smallest and fastest of the exhaustive search implementations.

BISECTION

The sum-of-odds method is clever. After optimization, it has a fast inner loop, but it doesn't scale well to large arguments. In this section, I present methods that have much more consistent timing and that scale well: bisection algorithms.

Bisection continually divides the search space in half, producing 1 bit of the result for each iteration. Therefore, only eight iterations are required to generate the square root of a 16-bit number, compared with up to 255 iterations for an exhaustive search. For 32-bit arguments, bisection requires 16 iterations, compared with up to 65,535 iterations

a) <pre>uint16_t bisect_multiply (uint16_t A) { uint8_t bit = 0x80; uint8_t x = 0; while (bit != 0) { x += bit; if (((uint16_t)x * x) > A) { x -= bit; } bit >>= 1; } return (x); }</pre>	b) <pre>uint16_t bisect_shiftsub (uint16_t A) { uint8_t bnum = 7; uint8_t x = 0; uint16_t x_sq = 0; uint16_t trial; while (bnum != 0xFF) { trial = x_sq + (x << (bnum + 1)) + (1 << (2 * bnum)); if (trial <= A) { x_sq = trial; x += (1 << bnum); } -- bnum; } return (x); }</pre>	c) <pre>uint16_t bisect_shsub_opt (uint16_t A) { uint16_t x_sh = 0; uint16_t bit_sh = x4000; uint16_t trial; while (bit_sh != 0) { trial = x_sh + bit_sh; if (trial <= A) { A -= trial; x_sh = trial + bit_sh; } x_sh >>= 1; bit_sh >>= 2; } #if SORT_ROUND if (A > x_sh) ++x_sh; #endif return (x_sh); }</pre>
--	--	--

Figure 2—These functions use bisection to calculate the square root of a 16-bit number. Bisection algorithms divide the search space in half until they find the solution. You can use multiplication to compute the square of each trial (a). The shift-subtract method computes the square root of each successive trial (b). You can also use an optimized shift-subtract method for an improvement in performance (c).

for an exhaustive search. A drawback to bisection is that it's more complex than an exhaustive search.

As with the exhaustive-search algorithms, I begin with a simple algorithm that uses one multiplication per loop. I then replace the multiplication with more basic operations. I conclude with optimized implementations in C and assembly.

Figure 2a shows bisection using one multiplication per loop. It starts by setting bit 7 of x to 1 ($\text{bit} = 0x80$) and testing x^2 against A . If $x^2 > A$, it clears bit 7. It then does the same for bits 6 through 0, and finally terminates after bit becomes 0. During my tests, the algorithm's performance varied from poor to excellent. The variations depended on whether or not the processor supported hardware multiplication.

$$\begin{aligned}
 x^2 &= (x_p + 2^i)^2 \\
 x^2 &= x_p^2 + 2x_p 2^i + 2^{2i} \\
 x^2 &= x_p^2 + 2^{i+1}x_p + 2^{2i} \\
 x^2 &= x_p^2 + [x_p << (i + 1)] + (1 << 2i)
 \end{aligned}$$

Figure 3—A little algebra shows how the new value of x^2 can be calculated from the previous values, x_p , and x_p^2 . No multiplication is required, only addition and shifting.

The multiplication in Figure 2a can be removed. Each time through the loop, the function sets a single bit in x , which amounts to adding a power of 2. In Figure 2a, no variable indicates which power of 2. For now, assume variable i holds the power of 2 (the bit number) for each loop. During the first loop, $i = 7$. In subsequent loops, it's decremented down to 0. Each loop begins with $x += \text{bit}$, which is equivalent to $x += 2^i$.

Suppose you save the value of x^2 from loop to loop. This will be handy in the next step, which calculates the new x^2 and compares it with A . Now get ready for a little bit of algebra. If x_p is the previous value of x (so x_p^2 is the previous value of x^2), you can calculate the new x^2 with the equations in Figure 3.

That wasn't so bad, was it? After a few short steps, you've replaced the multiplication in Figure 2a with left shifts and additions. All you need to do is save the previous value of x^2 and keep track of which bit is being set. Figure 2b is a direct implementation of this principle. It uses the variable bnum to keep track of the bit number (corresponding to i) and x_sq to save the value of x^2 . You can see that the

computation of the intermediate trial variable exactly matches the final equation for x^2 . If the trial value doesn't exceed A , then the algorithm sets the bit in x and updates x^2 .

Figure 2b is inefficient because of all the unnecessary shifting. Figure 2c is an optimized algorithm called the shift-subtract algorithm because the main operations shift and subtract. (For now, ignore the three lines starting with #if Sqrt_ROUND.) The first optimization is to eliminate the x_sq variable and subtract $2^{4+i}x_p + 2^{2i}$ from A . The result is that A always contains the error between the original argument and x^2 . If A_0 is the original value passed to the function, then $A = A_0 - x^2$ after each loop.

The other optimization is to keep the preshifted values of $2^{4+i}x_p$ and 2^{2i} in the x_sh and bit_sh variables (x shifted and bit shifted). Figure 4 shows how this works. Even though there is no variable i in the function, each step is identified with an i value to make it easier to keep track. The bit_sh variable represents 2^{2i} , so it's initialized to $0x4000 = 2^{14}$. The x_sh variable represents x shifted left by 2^{4+i} . It's initialized to 0.

Figure 4 shows only the bits of interest at each step in the calculation. A starts out with all bits unknown (represented by lowercase "a"), but as the algorithm progresses, some of the bits are forced to 0. Take my word for it. If you're skeptical, check out Figure 5 for proof.

The bit_sh variable has a single 1 bit, starting in bit 14, and shifted right by two positions each iteration. Finally, x_sh has 8 bits of interest that start in the high-order byte and are shifted right by one position for each iteration, ultimately ending in the low byte of the result. Each iteration generates 1 bit of x_sh (represented by a lowercase "x").

If you follow along in Figure 2c and Figure 4 simultaneously, you'll see that $(x_sh + bit_sh)$ is tentatively subtracted from A in the comparison ($trial \leq A$).

Remember that A has already had the previous value of x^2 subtracted from it, so this comparison is effectively testing the original A_0 against the new tentative value of x^2 . If the

i = 7	bit_sh	A
	x_sh	aaaaaaaaa aaaaaaaa
		1
		00000000
i = 6	bit_sh	aaaaaaaaa aaaaaaaa
	x_sh	1
		x0000000
		0
i = 5	bit_sh	0aaaaaaa aaaaaaaa
	x_sh	1
		xx000000
		00
i = 4	bit_sh	00aaaaaa aaaaaaaa
	x_sh	1
		xxx00000
		000
i = 3	bit_sh	000aaaaa aaaaaaaa
	x_sh	1
		xxxx0000
		0000
i = 2	bit_sh	0000aaaa aaaaaaaa
	x_sh	1
		xxxxxx000
		0000
i = 1	bit_sh	00000aaa aaaaaaaa
	x_sh	1
		xxxxxx00
		000000
i = 0	bit_sh	000000aa aaaaaaaa
	x_sh	1
		xxxxxxx0
		0000000
Final	bit_sh	0000000a aaaaaaaa
	x_sh	1
		xxxxxxx

Figure 4— i listed the variables in the shift-subtract algorithm at the beginning of each loop. Each iteration generates 1 bit of x and clears 1 bit of A . The i value is the bit number of x that's being generated. Variable A contains the error after each iteration.

comparison passes, then the algorithm subtracts the trial value from A and sets the bit in x_sh .

VARIATIONS

That completes my explanation of the shift-subtract algorithm, except for one variation. The algorithm as presented returns the truncated integer part of the square root. But what if you want to round the result instead of truncating it? The basic algorithm would return 49, which is $\sqrt{2,499}$. But 50 is a lot closer to the actual root. It turns out that you can use a simple test to round the result to the nearest integer.

You must round up to the next integer if the fractional part of the root is greater than 0.5. Round down otherwise. (It's impossible for the fraction

to be exactly 0.5 for the square root of an integer.) At the end of the shift-subtract algorithm, $A = A_0 - x^2$, which is greater than or equal to 0. If fractional part of $\sqrt{A_0}$ is greater than 0.5, then the following applies:

$$A_0 - (x + 0.5)^2 > 0$$

$$A_0 - x^2 - x - 0.25 > 0$$

$$A > x + 0.25$$

Because A and x are integers, the 0.25 term can be ignored. The rounding test is simply to round up if $A > x$. In Figure 2c, you can see this implemented at the end of the function. To round the result, define Sqrt_ROUND to be 1. To truncate it, make it 0.

If you need to deal with fixed-point numbers, you might be wondering how the shift-subtract algorithm works with them. Fixed-point numbers have an implied binary point, so they can represent numbers with an integer and a fractional part. They differ from floating-point numbers in that the position of the binary point doesn't vary at run-time.

One way to handle fixed-point numbers is to observe that the square root calculation divides the scale factor by two. So, if you prescale the input value to twice the desired output scale factor, the result will be correctly scaled. (This prescaling might require you to use a 32-bit square root function for 16-bit numbers.) There are shift-subtract variants that are tailored for fixed-point numbers, but I don't have space to cover those variants in this article.

Another well-known algorithm for computing square roots is the Newton-Raphson algorithm (also known as the divide-and-average method, or Newton's method), which is the method I learned on the calculator way back when. I've seen published code that uses this method for integer square roots, but I recommend against using it for this purpose for two reasons.

First, it's relatively slow on embedded processors that lack a hardware divide instruction. If you look back at the shift-subtract algorithm in Figure 2c, you can see that it resembles the classic binary division algorithm (in fact, the execution time is similar). The Newton-Raphson algorithm

$$A = err_i = A_0 - (x - \epsilon_i)^2 = A_0 - x^2 + 2x\epsilon_i - \epsilon_i^2$$

$$A = err_i = A_0 - A_0 + 2\epsilon_i\sqrt{A_0} - \epsilon_i^2 = 2\epsilon_i\sqrt{A_0} - \epsilon_i^2$$

$$A = err_i < 2(2^{i+1})2^8 - 2^{2(i+1)}$$

$$A = err_i < 2^{10+i} - 2^{2+2i}$$

Figure 5—The magnitude of A decreases as shown in Figure 3. err_i is the value of the error (A) at each step as i decreases from 7 to 0. A_0 is the initial value of A , which must be less than 2^{16} because it's a 16-bit variable. x is the final result, so $x^2 = A_0$, $x = \sqrt{A_0}$, and $x < 2^8$. Finally, ϵ_i is the error in x at each step, which is limited by the bits that have not been determined yet. Thus, $\epsilon_i < 2^{i+1}$.

Algorithm		Instruction words	Cycles		Normalized	
			Average	Maximum	Average	Maximum
Exhaustive search	SW Multiply	54*	20246	31719	121.96	176.22
	HW Multiply	16	1716	2558	10.34	14.21
	Sum-of-odds unoptimized	19	1888	2819	11.37	15.66
	Sum-of-odds optimized C	14	1379	2058	8.31	11.43
	Sum-of-odds optimized assembly	11	1208	1802	7.28	10.01
Bisection	SW Multiply	42*	1096	1138	6.60	6.32
	HW Multiply	16	108	108	0.65	0.60
	Shift-subtract unoptimized	63	869	952	5.23	5.29
	Shift-subtract optimized C	26	166	180	1.00	1.00
	Shift-subtract optimized assembly	25	152	166	0.92	0.92

Table 1—Study the timing results for all of the 16-bit integer square root algorithms. The results are normalized to the optimized C language shift-subtract algorithm. (* SW multiply instruction words don't include the size of the library routine used to perform the multiplication (12 words for the AVR GCC library).)

requires several divide operations, each of which takes nearly the same amount of time as the entire shift-subtract calculation.

The other problem with the Newton-Raphson algorithm is that it doesn't always converge. It sometimes gives a result that's off by one. I recommend against the method for integer square roots, although it can be a good algorithm for floating-point square roots.

RESULTS

I compiled each of the functions in Figures 1 and 2 using AVR GCC for the ATmega8 microcontroller, which supports hardware multiply. I also compiled for the AT90S8515 microcontroller, which doesn't support hardware multiply, forcing the compiler to use software multiplication. Finally, I created hand-optimized AVR assembly code for the sum-of-odds and shift-subtract methods.

Using a test program, I ran each func-

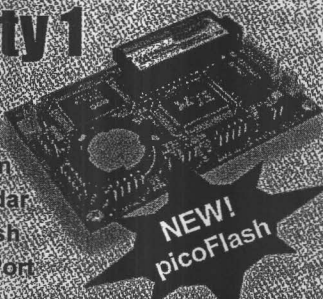
tion with every argument from 0 to 65535. I also recorded the maximum and average number of clock cycles. (This took several minutes to run on a 14.7456-MHz Atmel ATmega8 microcontroller.) The results are shown in Table 1. In addition to the number of cycles, I also included the number of 16-bit instruction words and the normalized number of cycles for each function. Normalization was achieved by dividing the cycles for each function by the cycles for the optimized C language implementation of the shift-subtract algorithm. Different compilers or processors would have had different results, but this test gives a good idea of relative performance.

The first thing to note is that the worst bisection algorithm is faster than the best exhaustive-search algorithm. A fast inner loop executed 255 times doesn't beat a slower inner loop executed only eight times.

The most surprising result is that the bisection algorithm with hardware multiply is the fastest of all. I really expected the optimized shift-subtract to be the fastest. But maybe this shouldn't be so

Embedded Ethernet only \$98 qty 1

- 10Base-T Ethernet
- 186 Processor @ 40 MHz
- DOS w/ Flash File System
- Hardware Clock / Calendar
- 512K DRAM & 512K Flash
- Console / Debug Serial Port
- 16 Digital I/O lines
- Optional DiskOnChip
- 5V DC Power
- Compact 3.75" x 2.50"



- (2) Serial Ports
- (2) 16-bit Timers
- Watchdog Timer

Expand with the pico-I/O

- 32 Digital Inputs
- 20 Digital Outputs
- 11 Channels of 12-Bit A/D
- C/C++ & Quick Basic Drivers

Call 530-297-6073 Email sales@jkmicro.com
On the web at www.jkmicro.com

JKmicrosystems

Connecting the World of Electronic Design

2006

CONFERENCE February 6-9, 2006

EXHIBITION February 7-8, 2006

Santa Clara Convention Center
Santa Clara, California

www.designcon.com

OFFICIAL SPONSOR

Agilent Technologies

DIAMOND SPONSOR

Rambus



International Engineering
Consortium
www.iec.org

surprising because the ATmega8 multiplies in two cycles. (The shift-subtract algorithm takes six cycles just to do the shifting.) A processor with a slower multiply instruction might give different results.

The 16-bit square root functions (in C) can be converted to 32 bits by doubling the widths of the argument, result, and intermediate values (i.e., change `uint8_t` and `uint16_t` to `uint16_t` and `uint32_t`). I created 32-bit sum-of-odds and shift-subtract functions and measured the execution time for an argument of 4,294,967,295 (the largest `uint32_t`). The sum-of-odds algorithm took 917,518 cycles. The shift-subtract algorithm took 670 cycles, a difference of more than three orders of magnitude!

So, now to the big question: Which algorithm is the best?

Based on the timings in Table 1, the bisection algorithms are the best choice for general use. If I had to choose one integer square root algorithm for my embedded tool kit, it would be the optimized shift-subtract algorithm in C because it works well even on processors without

hardware multiply. On a processor with hardware multiply, I'd consider the bisection algorithm using a multiplication per loop, but only after testing the execution time versus the shift-subtract method. For the fastest possible algorithm for a particular processor, I'd compare optimized assembly language versions of the multiply and shift-subtract algorithms.

The exhaustive-search algorithms did not perform well. Are there ever situations in which an exhaustive search is appropriate? Yes, but such cases are uncommon. The optimized sum-of-odds algorithm is the most compact function. If code size is critical and execution time doesn't matter, the sum-of-odds algorithm might be acceptable. Or consider an application that computes the square root of slowly changing input data. In this sort of case, start each new search at the previous root, and do only a small number of iterations per sample. You might find that a simple linear search outperforms bisection.

I hope you find this article useful the next time you need an algorithm to compute an integer square root. Even if you

don't need a square root algorithm, there are some useful conclusions to remember. The test results demonstrate that optimizing an inferior algorithm doesn't yield a superior solution. Sometimes what you think is an optimization is not. ■

Mike Dvorsky (bentbiker59-ccink@yahoo.com) has B.S. and M.S. degrees in electrical engineering from the University of Missouri-Rolla. After working for seven years in VLSI design and verification, Mike moved to the dark side and became an embedded software engineer. He's currently an embedded software lead engineer at a Fortune 500 company.

PROJECT FILES

To download the code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2006/187.

SOURCE

ATmega8 and AT90S8515 MCUs
Atmel Corp.
www.atmel.com

Actual Size

www.usbee.com

USBee AX

±10V Max
CH1 CH2

USBee AX-Standard comes with:

Digital Voltmeter Oscilloscope

16Msps, 100M+ Samples plus reference waveform
1-Channel, 2 Inputs, 8-bit, +/-10V input

Mixed Signal Oscilloscope

16Msps, 100M+ Samples
Analog: 1-Channel, 2-Inputs, 8-Bits Digital: 8 Channels

Logic Analyzer

8-Channel, 24Msps, 100M+ samples

CH1: 2.5 Volts
CH2: 0.1 Volts

AX-Plus adds:

- Digital Signal Generator
- USB Bus Decoder
- I2C Bus Decoder
- SPI Bus Decoder
- Async Bus Decoder

AX-Pro adds:

- Toolbuilder Source Code
- plus 7 more debug tools

USBee AX-Standard	\$445
USBee AX-Plus	\$745
USBee AX-Pro	\$1045

USBee

Test Pods

The USB-based Electrical Engineer

USBee.com (951)693-3065 support@usbee.com

Ideal for Students, Hobbyists and Engineers

Visit us today at www.USBee.com